

1. What is Kubernetes?

Kubernetes is a tool that helps us to run and manage applications in containers. It was developed by Google Lab in 2014, and it is also known as k8s. It is an open-source container orchestration platform that automates the deployment, management, and scaling of container-based applications in different kinds of environments like physical, virtual, and cloud-native computing foundations.

It is widely used in DevOps environments to manage complex application deployments and ensure they run consistently across different environments, from development to production.

2. How Docker and Kubernetes work together?

Docker is a platform that allows developers to create, deploy, and run applications inside containers. Containers package an application and its dependencies into a single unit that can run anywhere, ensuring consistency across multiple environments. Docker simplifies the development lifecycle by providing a standardized environment using local containers, which makes applications highly portable

Kubernetes, on the other hand, is a container orchestration tool that manages the deployment, scaling, and operation of containerized applications. It automates the scheduling and distribution of containers across a cluster of machines, ensuring that the desired state of the application is maintained. Kubernetes handles the complexities of managing multiple containers, such as load balancing, scaling, and ensuring high availability.

3. Why we need Kubernetes?

To understand why we need Kubernetes we need to first understand Containers. Once we have our application in various containers, we will now have to manage these Containers to ensure that the application is available to its users without any downtime. The key feature of containers is that they're small and light enough that we use them within our development environment. Using containers within our development environment gives us high confidence that our production environment is as similar as possible to that development environment. Here Kubernetes resolve this problem because Kubernetes is a container orchestration platform that automates the deployment, management, and scaling of container-based applications in different kinds of environments like physical, virtual, and cloud-native computing foundations.

4. Explain the features of Kubernetes?

Scalability – Kubernetes can automatically scale applications up or down based on demand. It can handle dynamic traffic patterns and adjust resources accordingly.

High Availability – It ensures that your applications are always available. Kubernetes can automatically restart containers that fail, replace containers, and distribute workloads across nodes to avoid downtime.

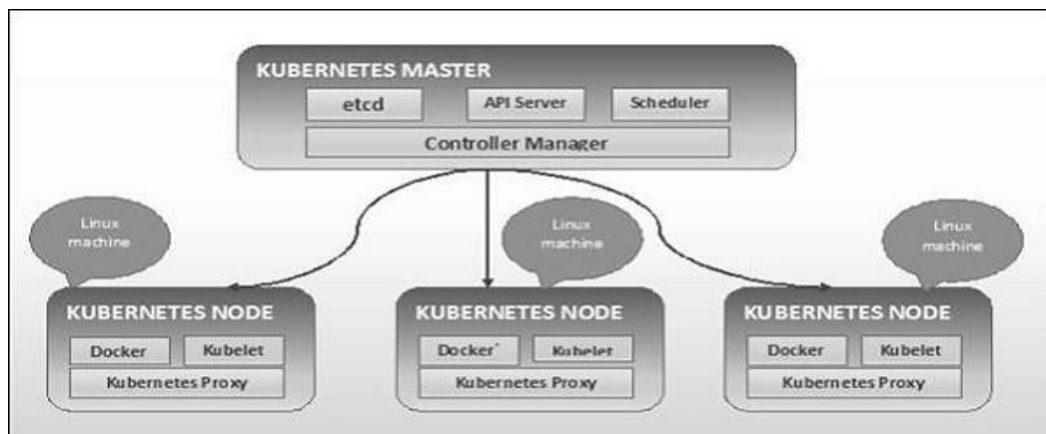
Load Balancing – Kubernetes has built-in load balancing to distribute traffic across containers effectively, which enhances application performance and reduces the risk of server overload.

Self-Healing – Kubernetes automatically detects failed containers and replaces or restarts them, ensuring that the system remains healthy and applications continue to run without manual intervention.

Portability – Kubernetes abstracts the underlying infrastructure, allowing applications to run consistently across different environments (on-premises, cloud, or hybrid).

Cost Efficiency – Kubernetes enables better resource utilization by allowing applications to run on shared resources and scale efficiently.

5. Explain about Kubernetes architecture and its components?



As seen in the following diagram, Kubernetes follows client-server architecture. Wherein, we have master installed on one machine and the node on separate Linux machines.

Kube-API server: The API server is a component of the Kubernetes control plane that exposes the Kubernetes API. It is like an initial gateway to the cluster that listens to updates or queries via CLI like Kubectl. Kubectl communicates with API Server to inform what needs to be done like creating pods or deleting pods etc. It also works as a gatekeeper. It generally validates requests received and then forwards them to other processes. No request can be directly passed to the cluster, it has to be passed through the API Server.

Kube-Scheduler: When API Server receives a request for Scheduling Pods then the request is passed on to the Scheduler. It intelligently decides on which node to schedule the pod for better efficiency of the cluster.

Kube-Controller-Manager: The kube-controller-manager is responsible for running the controllers that handle the various aspects of the cluster's control loop. These controllers include the replication controller, which ensures that the desired number of replicas of a given application is running, and the node controller, which ensures that nodes are correctly marked as "ready" or "not ready" based on their current state.

Etcd: It is a key-value store of a Cluster. The Cluster State Changes get stored in the etcd. It acts as the Cluster brain because it tells the Scheduler and other processes about which resources are available and about cluster state changes.

Node Components

These are the nodes where the actual work happens. Each Node can have multiple pods and pods have containers running inside them. There are 3 processes in every Node that are used to Schedule and manage those pods.

The following are the some of the components related to Node:

Container runtime: A container runtime is needed to run the application containers running on pods inside a pod. Example-> Docker

Kubelet: kubelet interacts with both the container runtime as well as the Node. It is the process responsible for starting a pod with a container inside.

kube-proxy: It is the process responsible for forwarding the request from Services to the pods. It has intelligent logic to forward the request to the right pod in the worker node.

Objects and Resources

Kubernetes manages everything in the cluster as objects. Key objects include:

Pods: The smallest deployable units that encapsulate one or more containers.

Services: Define how to expose pods to other services or external users.

ConfigMaps and Secrets: Store configuration and sensitive data separately from the application code.

Namespaces: Provide logical partitioning within a single cluster for resource isolation.

High-Level Workflow

- ✓ Users define the desired state of the system (e.g., applications, resources) through YAML/JSON files.
- ✓ The control plane(Master Node) interprets this and schedules the workloads on worker nodes.
- ✓ Worker nodes execute these workloads while reporting back their status to the control plane(Master Node).
- ✓ The control plane(Master Node) continuously monitors and maintains the desired state.

6. Explain about Kubernetes Jobs?

In the Kubernetes world, jobs are considered an object to act as a supervisor or controllers of a task. The Kubernetes job will create a pod, monitor the task, and recreate another one if that pod fails for some reason. Upon completion of the task, it will terminate the pod.

When you submit a job it will create one or more pods based on the requirement, will complete the task defined in the job, and keep the pods running until the task is completed. The Job keeps track of the successful completions when pods finish up. When a job is suspended, all of its active Pods are deleted until the job is restarted.

Types Of Jobs:

Simple Jobs: Executes a single task in one pod and completes after the task finishes.

Parallel Jobs: Runs multiple pods in parallel to complete a task. This is useful for tasks that can be divided into smaller, independent units of work.

CronJobs: A variation of Jobs that runs on a schedule (like a cron task). For example, you can schedule a backup job to run daily at a specific time.

Steps to create and execute the jobs:

Step 1. Start your minikube

```
$ minikube start
```

```
\vik $minikube start
🐳 minikube v1.27.1 on Darwin 12.6
🌟 Using the docker driver based on existing profile
👍 Starting control plane node minikube in cluster minikube
📦 Pulling base image ...
🔄 Restarting existing docker container for "minikube" ...
🔧 Preparing Kubernetes v1.25.2 on Docker 20.10.18 ...
🔍 Verifying Kubernetes components...
   ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟 Enabled addons: storage-provisioner, default-storageclass
🏁 Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```

Step 2. Create the job definition file in YAML format.

```
$ cat ping-job.yaml
```

```
\vik $cat ping-job.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: ping
spec:
  template:
    spec:
      containers:
      - image: busybox:latest
        command: ["ping", "geeksforgeeks.org"]
        imagePullPolicy: Always
        name: busybox
      restartPolicy: Never
      backoffLimit: 4
\vik $
```

Step 3. Submit the job definition to Kubectl, you should see the job created message on execution.

```
$ kubectl apply -f ping-job.yaml
```

```
\vik $
\vik $kubectl apply -f ping-job.yaml
job.batch/ping created
\vik $
```

Step 4. List the job using get jobs, we can see the no of completion of our job as well as the duration and age.

```
$ kubectl get jobs
```

```
\vik $kubectl apply -f ping-job.yaml
job.batch/ping created
\vik $
\vik $kubectl get job
NAME      COMPLETIONS  DURATION  AGE
ping      0/1          8s        8s
\vik $
```

Step 5. Get the job details.

```
$ kubectl describe job ping
```

```
vik $ kubectl describe job ping
Name:          ping
Namespace:     default
Selector:      controller-uid=ba5fec95-e8e9-4af0-8643-3ed2d1ed374c
Labels:        controller-uid=ba5fec95-e8e9-4af0-8643-3ed2d1ed374c
Annotations:   batch.kubernetes.io/job-tracking:
parallelism:   1
Completions:   1
Completion Mode: NonIndexed
Start Time:    Thu, 13 Oct 2022 00:24:29 -0700
Pods Statuses: 1 Active (1 Ready) / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  controller-uid=ba5fec95-e8e9-4af0-8643-3ed2d1ed374c
  job-name=ping
  Containers:
    busybox:
      Image:          busybox:latest
      Port:           <none>
      Host Port:      <none>
      Command:
        ping
        geeksforgeeks.org
      Environment:    <none>
      Mounts:         <none>
      Volumes:        <none>
Events:
  Type      Reason      Age    From      Message
  --      -
  Normal    SuccessfulCreate    21s    job-controller    Created pod_ping-h2xpj
```

Step 6. Get the pods running for our jobs, here you can see the pod name, how many containers, its status, whether its restarted or not, and the age of the pod.

```
$ kubectl get pods
```

```
vik $ kubectl get pod
NAME          READY   STATUS    RESTARTS   AGE
ping-h2xpj    1/1     Running   0           36s
vik $
vik $ kubectl describe pod ping-h2xpj
Name:          ping-h2xpj
Namespace:     default
Priority:       0
Service Account: default
Node:          minikube/192.168.49.2
Start Time:    Thu, 13 Oct 2022 00:24:29 -0700
Labels:        controller-uid=ba5fec95-e8e9-4af0-8643-3ed2d1ed374c
Annotations:   <none>
Status:        Running
IP:            172.17.0.3
IPs:           <none>
Controlled By: job/ping
Containers:
  busybox:
    Container ID:  docker://180545784fc273a80cf306dab66363c3ed27f
    Image:         docker-pullable://busybox@sha256:981896da5f712084e
    Image ID:      <none>
    Port:          <none>
    Host Port:     <none>
    Command:
      ping
      geeksforgeeks.org
```

To delete the job use kubectl delete command, If you delete the job, the pods associated with it will be deleted as well.

```
$ kubectl delete job ping
$ kubectl get job
```

```
\vik $ kubectl delete job ping
job.batch "ping" deleted
\vik $
\vik $ kubectl get job
No resources found in default namespace.
\vik $
\vik $ kubectl get pod
NAME          READY   STATUS    RESTARTS   AGE
ping-h2xpj    1/1     Terminating   0           14m
```

7. Explain about Kubernetes images?

Kubernetes (Docker) images are the key building blocks of Containerized Infrastructure. As of now, we are only supporting Kubernetes to support Docker images. Each container in a pod has its Docker image running inside it.

When we are configuring a pod, the image property in the configuration file has the same syntax as the Docker command does. The configuration file has a field to define the image name, which we are planning to pull from the registry.

```
apiVersion: v1
kind: pod
metadata:
  name: GeeksforGeeks
spec:
  containers:
    - name: gfg-image
      image: <docker_image_name>
      imagePullPolicy: Always
      command: ["echo", "SUCCESS"] ----->
```

8. Explain about Label and Selectors?

In Kubernetes, Labels and Selectors are mentioned on the configuration file of the deployments and services. They are used to connect kubernetes services with a kubernetes pod.

Labels are any key-value pairs that are used to identify that pod. The pod gets its label through the deployment which is like a blueprint for the pod before the pod is created. The Selector matches the label. Labels and selectors are required, to make connections between deployment, pods, and services.

The deployment is given labels in the format below:

```
"metadata":
{
  "labels":
  {
    "key1" : "value1",
    "key2" : "value2"
  }
}
```

The service identifies the pods and deployments using selectors in the format below:

```
"selector":
{
  "key" : "value"
}
```

9. Explain about Kubernetes yaml file tags?

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: example
  template:
    metadata:
      labels:
        app: example
    spec:
      containers:
        - name: example-container
          image: example-image
          ports:
            - containerPort: 8080
```

- **apiVersion:** Specifies the Kubernetes API version, such as "apps/v1" for Deployments.
- **kind:** Specifies the type of Kubernetes resource, in this case, "Deployment."
- **metadata:** Provides metadata for the Deployment, including the name, labels, and annotations.
- **spec:** Defines the desired state of the Deployment, including the number of replicas, the pod template, and any other related specifications. It includes:
 - **replicas:** Specifies the desired number of identical pod replicas to run.
 - **selector:** Specifies the labels that the Replica Set uses to select the pods it should manage.
 - **template:** Contains the pod template used for creating new pods, including container specifications, image names, and container ports.

10. Explain about Kubernetes name space?

Namespace provides an additional qualification to a resource name. This is helpful when multiple teams are using the same cluster and there is a potential of name collision. It can be as a virtual wall between multiple clusters.

Default Namespaces

Kubernetes comes with a few pre-defined namespaces:

1. **default:** Used for resources that do not explicitly specify a namespace.
2. **kube-system:** Contains system-related components, like the API server and scheduler.
3. **kube-public:** A special namespace for resources that should be publicly accessible across the cluster.
4. **kube-node-lease:** Used for node heartbeats and performance optimization.

You can create a custom namespace using YAML or the command line.

- **Using kubectl:**

Bash

```
kubectl create namespace my-namespace
```

- **Using a YAML File:**

Yaml

```
apiVersion: v1
kind: Namespace
metadata:
  name: my-namespace
```

11. Explain about Kubernetes node?

Kubernetes Nodes are the Worker or master machines where the actual work happens. Each Kubernetes node has the services required to execute Pods and is controlled by the Control Plane. Each Kubernetes Node can have multiple pods and pods have containers running inside them. 3 processes in every Node are used to Schedule and manage those pods.

12. Explain about Kubernetes Service DNS?

Service in Kubernetes will expose the application which is running in the pods to the internet. Each service will have its own set of endpoints from where they can be accessed from the internet.

For example, if you deployed a web application in the Kubernetes cluster with five replicas then you can access the five replicas with a single URL. The traffic will be routed to all the pods based on their incoming traffic. You can use ingress control to control the incoming traffic.

Example

```
apiVersion: v1
kind: Service
metadata:
  name: deployment-service
spec:
  selector:
    Tomcat: deploymentapp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

In the above we are exposing the Tomcat application deployed in the kubernetes cluster with labels of [Tomcat: deployment](#) app port: 80 is the port exposed internally in the cluster and port 8080 is the port of the [Tomcat](#) application.

13. Explain about multi port services in Kubernetes?

In Kubernetes, a **multi-port Service** allows you to define multiple ports for the same Service, enabling different types of traffic (e.g., HTTP, HTTPS, TCP) to be routed to the same set of Pods. This is especially useful if an application running in a Pod exposes multiple ports for different purposes, such as APIs, metrics, or administration interfaces.

Instead of creating multiple Services, you can define multiple ports in a single Service specification.

Example

```
apiVersion: v1
kind: Service
metadata:
  name: tomcat-service
spec:
  ports:
    - name: http
      port: 8080
      protocol: TCP
    - name: https
      port: 43
      protocol: TCP
```

By using the yaml which is mentioned above you can expose multiple ports but the IP address and DNS name will be the only one with that you can use various services.

14. Explain about Kubernetes node port service?

NodePort service in Kubernetes is a service that is used to expose the application to the internet from where the end-users can access it. If you create a NodePort Service Kubernetes will assign the port within the range of (30000-32767). The application can be accessed by end-users using the node's IP address.

Following is the sample YAML file for the kubernetes NodePort Service.

```
apiVersion: v1
kind: Service
metadata:
  name: <Name Of the Service>
spec:
  type: NodePort
  ports:
    - port: 80          # Port exposed within the cluster
      targetPort: 8080  # Port on the pods
      nodePort: 30000   # Port accessible externally on each node
  selector:
    app: example-app   # Select pods with this label
```

15. Difference between Service and Node Port?

Service:

- ✓ We can provide the internal & external service communication.
- ✓ Load Balancing is possible.
- ✓ Underlying Pods' IPs change dynamically.

Node Port:

- ✓ It will support external communication only.
- ✓ No load balancing
- ✓ It will use fixed set of IP addresses ranging from 30000-32767

Yaml

Copy

```
apiVersion: v1
kind: Service
metadata:
  name: my-nodeport-service
spec:
  type: NodePort
  selector:
    app: myapp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
      nodePort: 30001
```

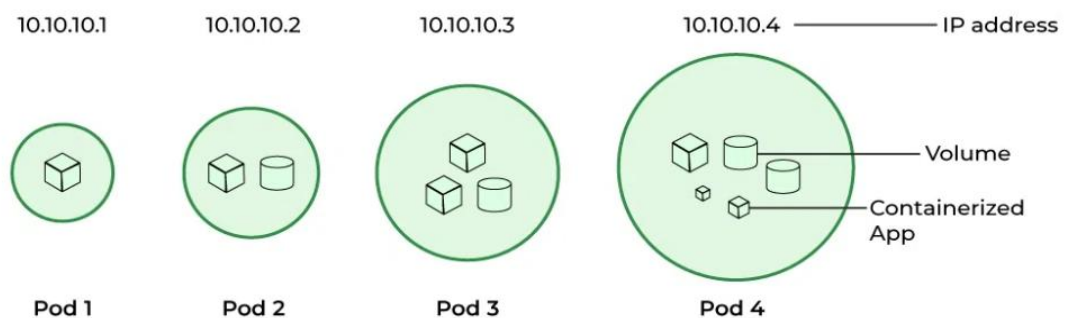
In this example:

- The Service listens on port `80` .
- It forwards traffic to Pods on port `8080` .
- The NodePort is `30001` , which can be accessed externally via `<NodeIP>:30001` .

16. Explain about pods in Kubernetes?

Pods in Kubernetes are like individual workers on a team. Each pod represents a specific task or process running in the cluster. They have their own unique address to communicate with other pods, storage space for saving data, and instructions on how to run their assigned job. While most pods have just one worker (container), some pods have a few workers that collaborate closely to get the job done efficiently.

A pod in a Kubernetes [cluster](#) indicates a process that is currently operating, and a pod may contain one or more containers. All of those containers share a single IP address, as well as the pod's storage, network, and any other requirements. A pod is a collection of one or more running containers, allowing for simple container movement within a cluster.



Types of Pods:

Single Container Pod: Pods in Kubernetes most often host a single container that provides all the necessary dependencies for an application to run. Single container pods are simple to create and offer a way for Kubernetes to control individual containers indirectly.

Multi Container Pod: Multi-container pods host containers that depend on each other and share the same resources. Inside such pods, containers can establish simple network connections and access the same storage volumes. Since they are all in the same pod, Kubernetes treats them as a single unit and simplifies their management.

Static Pod: Static pods in Kubernetes are like manually starting a program on your computer. You create a configuration file with details about the program you want to run and place it in a specific folder (usually /etc/kubernetes/manifests), and then your computer automatically starts running it without needing any extra commands. Similarly, in Kubernetes, you create a configuration file for a pod, place it in a designated folder on a node, and Kubernetes automatically starts running that pod on that node without needing to go through the usual Kubernetes control mechanisms.

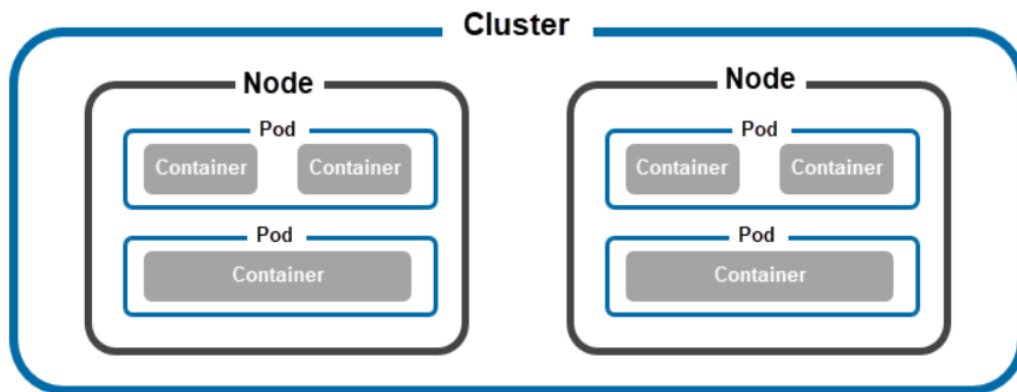
```
apiVersion: v1
kind: Pod
metadata:
  name: Tomcat
spec:
  containers:
    - name: Tomcat
      image: tomcat: 8.0
      ports:
        - containerPort: 7500
      imagePullPolicy: Always
    - name: Database
      image: mongoDB
      ports:
        - containerPort: 7501
      imagePullPolicy: Always
```

In the above code, we have created one pod with two containers inside it, one for tomcat and the other for MongoDB.

Troubleshooting with kubectl

- `kubectl get pods`: Lists all pods in the current namespace.
- `kubectl describe pod <pod-name>`: Provides detailed information about a specific pod.
- `kubectl logs <pod-name>`: Retrieves the logs from a specific pod.

17. Explain Cluster, Node and Pods?



Cluster: Cluster can contain a single node or collections of nodes

Node: A node can contain single pod or multiple pods

Pod: A pod can contains single container or multiple containers.

18. Difference between Cluster IP, Node Port and Load Balancer?

Feature	ClusterIP	NodePort	LoadBalancer
Accessibility	Accessible only within the cluster	Accessible externally via node's IP	Accessible externally via cloud load balancer
Use Cases	Internal communication between services	External access for development/testing	External access for production applications
Load Balancing	Load-balanced IP within the cluster	Each node forwards traffic on a specific port	External load balancer distributes traffic
IP Addresses	Single virtual IP address	Each node's IP address	Typically uses a public IP provided by cloud
Scaling	Scales horizontally with additional pods	Scales horizontally with additional nodes	Scales horizontally with additional nodes

19. Explain about replication controller?

Replication Controller is one of the key features of Kubernetes, which is responsible for managing the pod lifecycle. It is responsible for making sure that the specified number of pod replicas are running at any point of time. It is used in time when one wants to make sure

that the specified number of pod or at least one pod is running. It has the capability to bring up or down the specified no of pod.

It is a best practice to use the replication controller to manage the pod life cycle rather than creating a pod again and again.

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-rc
spec:
  replicas: 3
  selector:
    app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
        - name: nginx
          image: nginx:1.17
          ports:
            - containerPort: 80
```

- **replicas:** Specifies the desired number of Pods (3 in this case).
- **selector:** Identifies Pods that belong to this Replication Controller, based on their labels (`app: myapp`).
- **template:** Defines the Pod specification, including container images and configurations.

20. Explain about replica sets?

Replica Set ensures how many replica of pod should be running. It can be considered as a replacement of replication controller. The key difference between the replica set and the replication controller is, the replication controller only supports equality-based selector whereas the replica set supports set-based selector.

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: my-replicaset
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
        - name: nginx
          image: nginx:1.17
          ports:
            - containerPort: 80
```

- **replicas:** The desired number of Pod replicas (3 in this example).
- **selector:** Defines which Pods the ReplicaSet will manage, based on their labels (`app: myapp`).
- **template:** The Pod template specifies what new Pods should look like (e.g., container image, ports).
 - **metadata.labels:** Must match the `selector` to ensure Pods created by the ReplicaSet are properly managed.

21. Difference between Replication Controller and Replication Sets?

- The **Replication Controller** is largely considered outdated and is rarely used in modern Kubernetes setups.
- **ReplicaSets** are more flexible and are often managed indirectly through **Deployments**, which add advanced capabilities like rolling updates and rollbacks.

22. Explain about Kubernetes deployment?

Kubernetes deployment is sometimes taken to mean the general process of deploying applications on Kubernetes. A deployment allows you to describe an application's life cycle, such as which images to use for the app, the number of pods there should be, and the way in which they should be updated. You can scale the containers with the help of Kubernetes deployment up and down depending on the incoming traffic. If you have performed any rolling updates with the help of deployment and after some time if you find any bugs in it then you can perform rollback also. Kubernetes deployments are deployed with the help of CLI like [Kubectl](#) it can be installed on any platform.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
  labels:
    app: myapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
        - name: nginx
          image: nginx:1.17
          ports:
            - containerPort: 80
```

- **apiVersion**: Specifies the Kubernetes API version, such as "apps/v1" for Deployments.
- **kind**: Specifies the type of Kubernetes resource, in this case, "Deployment."
- **metadata**: Provides metadata for the Deployment, including the name, labels, and annotations.
- **spec**: Defines the desired state of the Deployment, including the number of replicas, the pod template, and any other related specifications. It includes:
 - **replicas**: Specifies the desired number of identical pod replicas to run.
 - **selector**: Specifies the labels that the Replica Set uses to select the pods it should manage.
 - **template**: Contains the pod template used for creating new pods, including container specifications, image names, and container ports.

Deployment Lifecycle

1. **Create**: Define the Deployment in YAML or use `kubectl apply`.
2. **Update**: Change the Deployment definition (e.g., update the container image version).
3. **Rolling Update**: Kubernetes gradually replaces old Pods with new Pods.
4. **Rollback (if needed)**: Revert to a previous configuration if the update fails.

23. Explain about ConfigMaps and secrets?

In Kubernetes, **ConfigMaps** and **Secrets** are resources used to manage configuration data and sensitive information for your applications. They decouple configuration settings from the application code, making it easier to manage and secure them.

ConfigMaps: Used to store non-sensitive configuration data in key-value pairs, such as environment variables, configuration files, or command-line arguments.

- **Example YAML for ConfigMap:**

```
Yaml

apiVersion: v1
kind: ConfigMap
metadata:
  name: example-configmap
data:
  DATABASE_URL: "mysql://example-user@example-db:3306/mydb"
  APP_ENV: "production"
```

- **Usage in a Pod:**

```
Yaml

spec:
  containers:
    - name: example-container
      image: nginx
      env:
        - name: DATABASE_URL
          valueFrom:
            configMapKeyRef:
              name: example-configmap
              key: DATABASE_URL
```

Secrets: Used to store sensitive information like passwords, API keys, and certificates securely. Data is stored in **base64-encoded format** (not encrypted by default, but you can integrate encryption at rest).

Example YAML for Secret:

```
Yaml

apiVersion: v1
kind: Secret
metadata:
  name: example-secret
type: Opaque
data:
  DATABASE_PASSWORD: bXktd2VjcmV0LXBhc3N3b3Jk
```

Usage in a Pod:

```
Yaml

spec:
  containers:
    - name: example-container
      image: nginx
      env:
        - name: DATABASE_PASSWORD
          valueFrom:
            secretKeyRef:
              name: example-secret
              key: DATABASE_PASSWORD
```

24. Explain about Kubernetes volumes?

In Kubernetes, a volume can be thought of as a directory which is accessible to the containers in a pod. We have different types of volumes in Kubernetes and the type defines how the volume is created and its content.

The concept of volume was present with the Docker, however the only issue was that the volume was very much limited to a particular pod. As soon as the life of a pod ended, the volume was also lost.

On the other hand, the volumes that are created through Kubernetes is not limited to any container. It supports any or all the containers deployed inside the pod of Kubernetes. A key advantage of Kubernetes volume is, it supports different kind of storage wherein the pod can use multiple of them at the same time.

Persistent Volume (PV) – It's a piece of network storage that has been provisioned by the administrator. It's a resource in the cluster which is independent of any individual pod that uses the PV.

Persistent Volume Claim (PVC) – The storage requested by Kubernetes for its pods is known as PVC. The user does not need to know the underlying provisioning. The claims must be created in the same namespace where the pod is created.

25. Explain about load balancing in Kubernetes?

Load balancing in Kubernetes is a critical feature that ensures efficient distribution of network traffic across multiple pods or services.

Internal Load Balancing: Managed within the Kubernetes cluster. Achieved using the **ClusterIP** service, which provides a stable internal IP address for communication between pods.

External Load Balancing: Exposes services to the outside world. Kubernetes offers several options:

NodePort: Opens a specific port on each node to forward traffic to the service.

LoadBalancer: Integrates with cloud provider load balancers to distribute traffic.

Ingress: Provides advanced routing capabilities, such as URL-based routing.

Service Discovery: Kubernetes uses the Service API to provide a stable IP or hostname for backend pods, ensuring seamless traffic routing even if pod instances change.

```

apiVersion: v1
kind: Service
metadata:
  name: my-loadbalancer-service
  labels:
    app: my-application
spec:
  type: LoadBalancer
  selector:
    app: my-application
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080

```

1. `type: LoadBalancer`: Indicates that Kubernetes should provision an external load balancer to direct traffic to this service.
2. `selector`: Matches the `app: my-application` label to identify the pods that should receive the traffic.
3. `ports`: Defines the ports:
 - `port`: The port on the Service exposed externally (e.g., 80).
 - `targetPort`: The port on the Pods that handle the traffic (e.g., 8080).

26. Explain about ingress in Kubernetes?

Ingress is a Kubernetes API object that is used to expose HTTP and HTTPS routes from outside the Kubernetes cluster to services inside the cluster. It provides a single entry point into a cluster hence making it simpler to manage applications and troubleshoot routing issues. The official definition of Ingress says **"Ingress is an API object that manages external access to the services in a Cluster"**.

An [Ingress controller](#) is responsible for fulfilling the Ingress, usually with a load balancer, though it may also configure your edge router or additional frontends to help handle the traffic. There are many different Ingress controllers, and there's support for cloud-native load balancers (from GCP, AWS, and Azure).

e.g. Nginx, Ambassador, EnRoute, HAProxy, AWS ALB, AKS Application Gateway

The Ingress [spec](#) has all the information needed to configure a load balancer or proxy server. It contains a list of rules matched against all incoming requests. Ingress provides routing rules to manage external users' access to the services in a Kubernetes cluster, typically via HTTPS/HTTP. With Ingress, you can easily set up rules for routing traffic without creating a bunch of Load Balancers or exposing each service on the node. This makes it the best option to use in production environments.


```

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: example-service
            port:
              number: 80

```

27. Explain about Kubernetes auto scaling?

Kubernetes auto-scaling is a feature that adjusts the number of pods, or even the number of nodes in a cluster, based on workload demands. This ensures efficient resource utilization and helps maintain optimal performance. There are two primary types of auto-scaling in Kubernetes:

Horizontal Pod Autoscaler (HPA): Automatically scales the number of pods in a deployment, replica set, or stateful set. It uses CPU/memory utilization or custom metrics as triggers.

```

apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: example-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: example-deployment
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 80

```

- `minReplicas` and `maxReplicas`: The minimum and maximum number of pods.
- **Metric target:** In this case, it scales when the CPU usage exceeds 80%.

Cluster Autoscaler: Dynamically adjusts the number of nodes in the cluster. Removes underutilized nodes or adds nodes to handle increased workloads. Typically works with cloud providers like AWS, Azure, and GCP. Configuration is done at the cluster level, and it requires proper setup in the cloud environment.

28. Explain about kubectl?

kubectl is the command-line tool used to interact with Kubernetes clusters. It acts as a bridge between you and the Kubernetes control plane(Master Node), allowing you to manage and automate tasks, such as deploying applications, inspecting resources, scaling workloads, and troubleshooting issues within the cluster.

Common Commands:

- Get Information:**

Bash

kubectl get pods

kubectl get services

Copy
- Describe Resources:**

Bash

kubectl describe pod <pod-name>

Copy
- Create/Apply Resources:**

Bash

kubectl apply -f <file.yaml>

Copy
- Delete Resources:**

Bash

kubectl delete pod <pod-name>

Copy
- Access Pod Logs:**

Bash

kubectl logs <pod-name>

Copy
- Debugging (Exec Into Pod):**

Bash

kubectl exec -it <pod-name> -- /bin/bash

Copy

Command	Description
kubectl get pods	Lists all pods in the current namespace.
kubectl get pods --all-namespaces	Lists all pods across all namespaces.
kubectl get all	Displays all Kubernetes objects in the current namespace.
kubectl create deployment <name> --image=<image>	Creates a new deployment with the specified name and image.
kubectl delete pod <pod_name>	Deletes a specific pod by its name.
kubectl describe pod <pod_name>	Shows detailed information about a specific pod.
kubectl logs <pod_name>	Allows you to view logs for a specific pod.
kubectl exec -it <pod_name> -- /bin/bash	Executes a command inside a running pod (e.g., opens a bash shell).
kubectl scale deployment <deployment_name> -- replicas=<number>	Scales a deployment to a specified number of replicas.
kubectl port-forward <pod_name> <local_port>: <pod_port>	Forwards a local port to a port on a pod for easier access.

<code>kubectl apply -f <filename>.yaml</code>	Creates or updates resources defined in a YAML file.
<code>kubectl get nodes</code>	Lists all nodes in the cluster.
<code>kubectl describe node <node_name></code>	Displays detailed information about a specific node.
<code>kubectl get namespaces</code>	Lists all namespaces in the cluster.
<code>kubectl config use-context <context_name></code>	Switches to a specific context defined in your kubeconfig file.
<code>kubectl version</code>	Shows the version information of the kubectl client and the Kubernetes server.
<code>kubectl get svc</code>	Lists all services in the current namespace.
<code>kubectl get deployments</code>	Displays all deployments in the current namespace.
<code>kubectl get replicaset</code>	Lists all ReplicaSets in the current namespace.
<code>kubectl get daemonsets</code>	Lists all DaemonSets in the current namespace.
<code>kubectl get statefulsets</code>	Lists all StatefulSets in the current namespace.
<code>kubectl edit deployment <deployment_name></code>	Opens an existing deployment for editing in your default editor.
<code>kubectl rollout status deployment/<deployment_name></code>	Checks the status of a deployment rollout.
<code>kubectl rollout undo deployment/<deployment_name></code>	Rolls back to the previous version of a deployment.
<code>kubectl get configmaps</code>	Lists all ConfigMaps in the current namespace.
<code>kubectl get serviceaccounts</code>	Displays all service accounts in the current namespace.
<code>kubectl top pods</code>	Shows metrics for pods, including CPU and memory usage.
<code>kubectl top nodes</code>	Displays metrics for nodes, including CPU and memory usage.
<code>kubectl get persistentvolumes</code>	Lists all PersistentVolumes in the cluster.

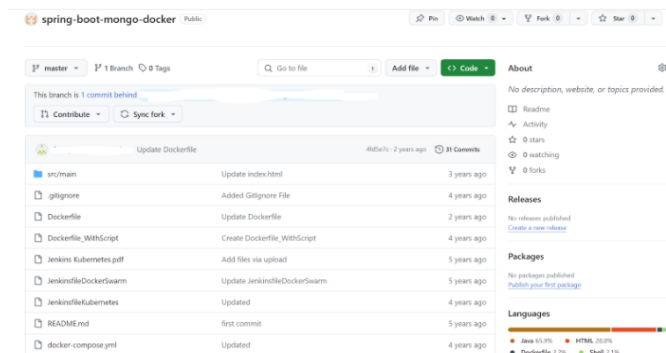
29. Explain the steps to deploy the dockized spring boot application in Kubernetes?

Steps for Deployment

1. Write and build the Java Spring Boot Application
2. Containerize the Java Spring Boot Application
3. Push the container to a Container Registry such as DockerHub
4. Create the required YAML Manifest for Kubernetes Resources
5. Apply the YAML manifest to the Kubernetes clusters

Step 1: Create the [GitHub repository](#).

Step 2: Here is my spring boot GitHub repository.



Step 3: Create a Dockerfile

In the root of the application directory, create a file named `Dockerfile` with the following content:

```
FROM openjdk:8-jdk-alpine
EXPOSE 8080
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

Step 4: Build the Docker Image

Build the Docker image with the following command:

```
$ docker build -t my-spring-boot-app:latest .
```

Step 5: Push the Docker Image to a Registry

Push the Docker image to a registry such as Docker Hub, Google Container Registry, or Amazon Elastic Container Registry. Replace `<your-dockerhub-username>` with your Docker Hub username.

```
$ docker tag my-spring-boot-app:latest <your-dockerhub-username>/my-spring-boot-
$ docker push <your-dockerhub-username>/my-spring-boot-app:latest
```

Step 6: Create a Kubernetes Deployment

Create a file named `deployment.yaml` with the following content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-spring-boot-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-spring-boot-app
  template:
    metadata:
      labels:
        app: my-spring-boot-app
    spec:
      containers:
        - name: my-spring-boot-app
          image: <your-dockerhub-username>/my-spring-boot-app:latest
          ports:
            - containerPort: 8080
```

This configuration defines a Kubernetes Deployment with three replicas of the Spring Boot application.

Step 7: Create a Kubernetes Service

Create a file named `service.yaml` with the following content:

```
apiVersion: v1
kind: Service
metadata:
  name: my-spring-boot-app
spec:
  selector:
    app: my-spring-boot-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  type: LoadBalancer
```

This configuration defines a Kubernetes Service that exposes the Spring Boot application on port 80 and load balances traffic between the replicas.

Step 8: Deploy to Kubernetes

Apply the Kubernetes configurations to your cluster with the following commands:

```
$ kubectl apply -f deployment.yaml
$ kubectl apply -f service.yaml
```

Step 9: Access the Application

Wait for the external IP address to be assigned to the service. Check the service's external IP with the following command:

```
$ kubectl get svc my-spring-boot-app
```

Once the external IP is available, access the application using the IP address and port 80 in your browser or any HTTP client.

```
kubectl get svc
```

```
kubernetes  ClusterIP  10.96.0.1      <none>      443/TCP      7d12h
mongosvc    ClusterIP  10.100.206.226 <none>      80/TCP       14m
springsvc   NodePort   10.111.136.72  <none>      80:30259/TCP 14m
controlplane $
```

30. Explain the steps to dockerised spring boot app in Kubernetes cluster using Jenkins?

1. Prepare Your Spring Boot Application

- Ensure your Spring Boot application is ready and has a `Dockerfile` to containerize it.
- Example `Dockerfile`:

Dockerfile

 Copy

```
FROM openjdk:11-jre-slim
COPY target/app.jar app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

2. Set Up Jenkins


- Install Jenkins on your server or use a cloud-hosted Jenkins instance.
- Install necessary plugins:
 - Docker Pipeline Plugin
 - Kubernetes Continuous Deploy Plugin
 - Git Plugin
- Configure Jenkins to connect to your Docker daemon and Kubernetes cluster.

3. Create a Jenkins Pipeline

- Write a `Jenkinsfile` to automate the CI/CD process. Example stages:

1. Clone the Repository:

Groovy

 Copy

```
stage('Clone Repository') {
    steps {
        git 'https://github.com/your-repo.git'
    }
}
```

2. Build and Dockerize:


Groovy

 Copy

```
stage('Build & Dockerize') {
    steps {
        sh 'mvn clean package'
        sh 'docker build -t your-dockerhub-repo/app:latest .'
    }
}
```

3. Push Docker Image:


Groovy

 Copy

```
stage('Push Image') {
    steps {
        sh 'docker login -u your-username -p your-password'
        sh 'docker push your-dockerhub-repo/app:latest'
    }
}
```

4. Deploy to Kubernetes:

Groovy

 Copy

```
stage('Deploy to Kubernetes') {
    steps {
        sh 'kubectl apply -f k8s-deployment.yaml'
    }
}
```

4. Create Kubernetes Deployment Files

- Example `k8s-deployment.yaml`:

Yaml

 Copy

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: springboot-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: springboot-app
  template:
    metadata:
      labels:
        app: springboot-app
    spec:
      containers:
        - name: springboot-app
          image: your-dockerhub-repo/app:latest
          ports:
            - containerPort: 8080
```

```
---
apiVersion: v1
kind: Service
metadata:
  name: springboot-service
spec:
  type: LoadBalancer
  selector:
    app: springboot-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

5. Run the Pipeline

- Trigger the Jenkins pipeline to:
 1. Build the application.
 2. Create a Docker image.
 3. Push the image to Docker Hub.
 4. Deploy the application to your Kubernetes cluster.